1. **Procedure to register a new use**

DELIMITER //

CREATE DEFINER=`root`@`localhost` PROCEDURE `register\_user`(

IN name VARCHAR(100),

IN email VARCHAR(100),

IN password VARCHAR(255),

IN location VARCHAR(100),

IN profile\_picture VARCHAR(255),

IN bio TEXT,

IN verified\_status TINYINT(1),

IN tokens INT,

IN event\_count INT,

IN status VARCHAR(20)

)

BEGIN

INSERT INTO `user` (

`name`, `email`, `password`, `location`, `profile\_picture`,

`bio`, `verified\_status`, `tokens`, `event\_count`, `status`

)

VALUES (

name, email, password, location, profile\_picture,

bio, verified\_status, tokens, event\_count, status

);

END; //

DELIMITER ;

CALL register\_user(

'Sara',

'sara@example.com',

'hashedpassword456',

NULL,

NULL,

NULL,

0,

0,

NULL,

'Inactive'

);

1. **Procedure to deactivate inactive users**

DELIMITER //

CREATE PROCEDURE deactivate\_inactive\_users()

BEGIN

UPDATE user

SET status = 'Inactive'

WHERE last\_login < DATE\_SUB(NOW(), INTERVAL 1 YEAR);

END;

//

DELIMITER ;

1. **View for active users**

CREATE VIEW active\_users AS SELECT user\_id, name, email FROM users WHERE status = 'Active';

1. **Query to get all active users**

SELECT \* FROM active\_users;

1. **Procedure to get attendance report**

DELIMITER //

CREATE PROCEDURE get\_attendance\_report(

IN start\_date DATE,

IN end\_date DATE

)

BEGIN

SELECT a.attendance\_id, a.event\_id, a.user\_id, a.status, e.title

FROM attendance a

JOIN event e ON a.event\_id = e.event\_id

WHERE e.start\_time BETWEEN start\_time AND end\_time;

END;

//

DELIMITER ;

1. **Call**:  
   CALL get\_attendance\_report('2024-12-01', '2024-12-04');
2. **View for events with attendee count**

CREATE VIEW event\_attendee\_count AS

SELECT e.event\_id, e.title, COUNT(a.attendance\_id) AS attendee\_count

FROM event e

LEFT JOIN attendance a ON e.event\_id = a.event\_id

GROUP BY e.event\_id, e.title;

1. **Trigger for token purchase**

DELIMITER //

CREATE TRIGGER after\_token\_purchase

AFTER INSERT ON transactions

FOR EACH ROW

BEGIN

-- Check if the transaction type is 'Purchase'

IF NEW.transaction\_type = 'Purchase' THEN

-- Update the user's tokens in the `user` table

UPDATE `user`

SET tokens = tokens + NEW.tokens

WHERE user\_id = NEW.user\_id;

END IF;

END;

//

DELIMITER ;

Use:

INSERT INTO transactions (user\_id, transaction\_type, tokens, amount)

VALUES (1, 'Purchase', 50, 500.00);

1. **Trigger to log token deductions**

DELIMITER //

CREATE TRIGGER after\_course\_purchase

AFTER INSERT ON course\_purchases

FOR EACH ROW

BEGIN

-- Check if the user has enough tokens to purchase the course

IF (SELECT tokens FROM `user` WHERE user\_id = NEW.user\_id) >= NEW.tokens\_required THEN

-- Deduct the tokens from the user's account

UPDATE `user`

SET tokens = tokens - NEW.tokens\_required

WHERE user\_id = NEW.user\_id;

ELSE

-- Raise an error if the user does not have enough tokens

SIGNAL SQLSTATE '45000'

SET MESSAGE\_TEXT = 'Insufficient tokens for course purchase.';

END IF;

END;

//

DELIMITER ;

**Test:**

INSERT INTO course\_purchases (user\_id, course\_id, tokens\_required)

VALUES (1, 5, 30);

1. **User token balance view**

CREATE VIEW user\_token\_balance AS

SELECT u.user\_id, u.name, u.tokens, t.transaction\_type, t.tokens AS token\_change, t.amount, t.transaction\_date

FROM user u

LEFT JOIN transactions t ON u.user\_id = t.user\_id

ORDER BY t.transaction\_date DESC;

1. **View for course earnings**

CREATE VIEW course\_earnings AS

SELECT c.course\_name, c.tokens\_earned, u.name AS uploader, c.upload\_date

FROM courses c

JOIN user u ON c.user\_id = u.user\_id;

1. **Procedure to buy tokens**

DELIMITER //

CREATE PROCEDURE buy\_tokens(IN userId INT, IN tokenCount INT, IN amountPaid DECIMAL(10, 2))

BEGIN

UPDATE user

SET tokens = tokens + tokenCount

WHERE user\_id = userId;

INSERT INTO transactions (user\_id, transaction\_type, tokens, amount)

VALUES (userId, 'Purchase', tokenCount, amountPaid);

END //

DELIMITER ;

1. **Call**

CALL buy\_tokens(1, 50, 500.00);

1. **Procedure to upload a course and earn tokens**

DELIMITER //

CREATE PROCEDURE upload\_course(IN userId INT, IN courseName VARCHAR(255), IN courseDesc TEXT, IN tokenReward INT)

BEGIN

INSERT INTO courses (user\_id, course\_name, course\_description, tokens\_earned)

VALUES (userId, courseName, courseDesc, tokenReward);

UPDATE users

SET tokens = tokens + tokenReward

WHERE user\_id = userId;

INSERT INTO transactions (user\_id, transaction\_type, tokens)

VALUES (userId, 'Earned', tokenReward);

END //

DELIMITER ;

1. **Procedure to access a course**

DELIMITER //

CREATE PROCEDURE access\_course(IN userId INT, IN tokensRequired INT)

BEGIN

UPDATE users

SET tokens = tokens - tokensRequired

WHERE user\_id = userId;

INSERT INTO transactions (user\_id, transaction\_type, tokens)

VALUES (userId, 'Deducted', -tokensRequired);

END //

DELIMITER ;

END //

DELIMITER ;

1. **Procedures To find courses**

DELIMITER //  
CREATE PROCEDURE find\_courses\_by\_title(  
    IN search\_title VARCHAR(255)  
)  
BEGIN  
    SELECT course\_id, course\_name, course\_description, tokens\_earned, upload\_date, user\_id  
    FROM courses  
    WHERE course\_name LIKE CONCAT('%', search\_title, '%')  
    ORDER BY course\_name;  
END;  
//  
DELIMITER ;

**Call :** CALL find\_courses\_by\_title('Python');

1. **Messege store process**

DELIMITER //  
CREATE PROCEDURE send\_message(  
    IN sender INT,  
    IN receiver INT,  
    IN message TEXT  
)  
BEGIN  
    -- Insert the message into the messages table  
    INSERT INTO messages (sender\_id, receiver\_id, message\_text)  
    VALUES (sender, receiver, message);  
  
    -- Optional: You could log or perform additional actions here if needed  
END;  
//  
DELIMITER ;

Call: CALL send\_message(1, 2, 'Hello, how are you?');

Call: CALL send\_message(3, 5, 'Don’t forget to check the new course!');  
  
**18. Procedure To View messege**

CREATE VIEW message\_view AS  
SELECT   
    m.message\_id,  
    m.sender\_id,  
    [s.name](https://s.name/) AS sender\_name,  
    m.receiver\_id,  
    [r.name](https://r.name/) AS receiver\_name,  
    m.message\_text,  
    m.sent\_at,  
    m.is\_read  
FROM   
    messages m  
JOIN   
    user s ON m.sender\_id = s.user\_id  
JOIN   
    user r ON m.receiver\_id = r.user\_id  
ORDER BY   
    m.sent\_at DESC;02:19 AM

1. **Messege Table:**

SELECT \* FROM message\_view;

1. **View message for specific user**

-- Messages where user\_id = 1 is the sender or receiver  
SELECT \*   
FROM message\_view  
WHERE sender\_id = 1 OR receiver\_id = 1;

1. Procedure To read messege:

DELIMITER //  
CREATE PROCEDURE read\_messages(  
    IN receiver\_id INT  
)  
BEGIN  
    -- Update all unread messages for the receiver to is\_read = 1  
    UPDATE messages  
    SET is\_read = 1  
    WHERE receiver\_id = receiver\_id AND is\_read = 0;  
  
    -- Retrieve the updated messages for the receiver  
    SELECT message\_id, sender\_id, receiver\_id, message\_text, sent\_at, is\_read  
    FROM messages  
    WHERE receiver\_id = receiver\_id  
    ORDER BY sent\_at DESC;  
END;  
//  
DELIMITER ;

**Call:**

CALL read\_messages(2);

1. **Procedure to give course Review**

DELIMITER //  
CREATE PROCEDURE give\_course\_review(  
    IN user\_id INT,  
    IN course\_id INT,  
    IN rating INT,  
    IN review\_text TEXT  
)  
BEGIN  
    -- Check if the user is eligible to review (e.g., completed the course)  
    IF EXISTS (  
        SELECT 1  
        FROM transactions  
        WHERE user\_id = user\_id AND transaction\_type = 'Earned' AND course\_id = course\_id  
    ) THEN  
        -- Insert the review into the reviews table  
        INSERT INTO reviews (user\_id, course\_id, rating, review\_text)  
        VALUES (user\_id, course\_id, rating, review\_text);  
    ELSE  
        SIGNAL SQLSTATE '45000'  
        SET MESSAGE\_TEXT = 'User is not eligible to review this course';  
    END IF;  
END;  
//  
DELIMITER ;  
**Call:**   
CALL give\_course\_review(1, 3, 5, 'Excellent course! Very informative.');

CALL give\_course\_review(2, 5, 4, 'Good course, but could be improved.');

1. **Procedure to update ranking**

DELIMITER //  
CREATE PROCEDURE update\_user\_ranking()  
BEGIN  
    -- Loop through all users to calculate ranking scores  
    DECLARE done INT DEFAULT FALSE;  
    DECLARE userId INT;  
    DECLARE userCursor CURSOR FOR SELECT user\_id FROM user;  
    DECLARE CONTINUE HANDLER FOR NOT FOUND SET done = TRUE;  
  
    OPEN userCursor;  
  
    userLoop: LOOP  
        FETCH userCursor INTO userId;  
        IF done THEN  
            LEAVE userLoop;  
        END IF;  
  
        -- Calculate ranking score based on reviews  
        SET @positive\_score = (  
            SELECT COALESCE(AVG(rating) \* 10, 0) -- Average rating multiplied by 10  
            FROM reviews  
            WHERE user\_id = userId  
        );  
  
        -- Calculate penalty based on reports  
        SET @negative\_score = (  
            SELECT COUNT(\*)  
            FROM reports  
            WHERE reported\_user\_id = userId  
        );  
  
        -- Update the ranking score  
        SET @final\_score = GREATEST(@positive\_score - (@negative\_score \* 5), 0); -- Deduct 5 points per report, minimum score is 0  
  
        -- Update the user's ranking score in the user table  
        UPDATE user  
        SET ranking\_score = @final\_score  
        WHERE user\_id = userId;  
  
        -- Assign rank based on the ranking score  
        UPDATE user  
        SET rank = CASE  
            WHEN ranking\_score >= 300 THEN 'Expert'  
            WHEN ranking\_score >= 200 THEN 'Advanced'  
            WHEN ranking\_score >= 100 THEN 'Intermediate'  
            ELSE 'Beginner'  
        END  
        WHERE user\_id = userId;  
    END LOOP;  
  
    CLOSE userCursor;  
END;  
//  
DELIMITER ;

**Call:**

CALL update\_user\_ranking();  
  
**24. View users according to ranking score**

SELECT \* FROM user\_ranking\_view;

SELECT \*   
FROM user\_ranking\_view  
LIMIT 5;

SELECT \*   
FROM user\_ranking\_view  
WHERE rank = 'Expert';